Constructing LZ78 Tries and Position Heaps in Linear Time for Large Alphabets

Yuto Nakashima, Tomohiro I, Shunsuke Inenaga,
Hideo Bannai, and Masayuki Takeda
Department of Informatics, Kyushu University, Fukuoka 819-0395, Japan
{yuto.nakashima,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp
tomohiro.i@cs.tu-dortmund.de

Abstract

We present the first worst-case linear-time algorithm to compute the Lempel-Ziv 78 factorization of a given string over an integer alphabet. Our algorithm is based on nearest marked ancestor queries on the suffix tree of the given string. We also show that the same technique can be used to construct the position heap of a set of strings in worst-case linear time, when the set of strings is given as a trie.

1 Introduction

Lempel-Ziv 78 (*LZ78*, in short) is a well known compression algorithm [19]. LZ78 compresses a given text based on a dynamic dictionary which is constructed by partitioning the input string, the process of which is called LZ78 factorization. Other than its obvious use for compression, the LZ78 factorization is an important concept used in various string processing algorithms and applications [6, 13].

In this paper, we show an LZ78 factorization algorithm which runs in $O(n)$ time using $O(n)$ working space for an integer alphabet, where n is the length of a given string and m is the size of the LZ78 factorization. Our algorithm does not make use of any randomization such as hashing, and works in $O(n)$ time *in the worst case*. To our knowledge, this is the first $O(n)$-time LZ78 factorization algorithm when the size of an integer alphabet is $O(n)$ and $2^{\omega(\log n \frac{\log\log\log n}{(\log\log n)^2})}$. Our algorithm computes the LZ78 trie (a trie representing the LZ78 factors) via the suffix tree [17] annotated with a semi-dynamic nearest marked ancestor data structure [18, 1].

We also show that the same idea can be used to construct the *position heap* [7] of a set of strings which is given as a trie, and present an $O(\ell)$-time algorithm to construct it, where ℓ is the size of the given trie.

Some of the results of this paper appeared in the preliminary versions [14, 2].

Comparison to previous work

The LZ78 trie (and hence the LZ78 factorization) of a string of length n can be computed in $O(n)$ expected time and $O(m)$ space, if hashing is used for maintaining the branching nodes of the LZ8 trie [10]. In this paper, we focus on algorithms without randomization, and we are interested in the worst-case behavior of LZ78 factorization algorithms. If balanced binary search trees are used in place of hashing, then the LZ78 trie can be computed in $O(n \log \sigma)$ worst-case time and $O(m)$ working space. Our $O(n)$-time algorithm is faster than this method when $\sigma \in \omega(1)$ and $\sigma \in O(n)$. On the other hand, our algorithm uses $O(n)$ working space, which can be larger than $O(m)$ when the string is LZ8 compressible. Jansson et al. [12] proposed an algorithm which

computes the LZ78 trie of a given string in $O(n(\log\log n)^2/(\log_\sigma n \log\log\log n))$ worst-case time, using $O(n(\log\sigma + \log\log_\sigma n)/\log_\sigma n)$ bits of working space. Our $O(n)$-time algorithm is faster than theirs when $\sigma \in 2^{\omega(\log n \frac{\log\log\log n}{(\log\log n)^2})}$ and $\sigma \in O(n)$, and is as space-efficient as theirs when $\sigma \in \Theta(n)$. Tamakoshi et al. [16] proposed an algorithm which computes the LZ78 trie in $O(n + (s+m)\log\sigma)$ worst-case time and $O(m)$ working space, where s is the size of the run length encoding (RLE) of a given string. Our $O(n)$-time algorithm is faster than theirs when $\sigma \in 2^{\omega(\frac{n}{s+m})}$ and $\sigma \in O(n)$.

The position heap of a single string of length n over an alphabet of size σ can be computed in $O(n\log\sigma)$ worst-case time and $O(n)$ space [7], if the branches in the position heap are maintained by balanced binary search trees. Independently of this present work, Gagie et al. [11] showed that the position heap of a given string of length n over an integer alphabet can be computed in $O(n)$ time and $O(n)$ space, via the suffix tree of the string.

2 Preliminaries

2.1 Notations on strings

We consider a string w of length n over integer alphabet $\Sigma = \{1, \ldots, \sigma\}$, where $\sigma \in O(n)$. The length of w is denoted by $|w|$, namely, $|w| = n$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The set of suffixes of a string w is denoted by *Suffix*(w). The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq n$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq n$. For convenience, let $w[i..j] = \varepsilon$ if $j < i$. For any string w, let w^R denote the reversed string of w, i.e., $w^R = w[n]w[n-1]\cdots w[1]$.

2.2 Suffix Trees

We give the definition of a very important and well known string index structure, the suffix tree. To assure property 4 below for the sake of presentation, we assume that string w ends with a unique character that does not occur elsewhere in w.

Definition 1 (Suffix Trees [17])**.** *For any string w, its suffix tree, denoted STree(w), is a labeled rooted tree which satisfies the following:*

1. *each edge is labeled with a non-empty substring of w;*
2. *each internal node has at least two children;*
3. *the labels x and y of any two distinct out-going edges from the same node begin with different symbols in Σ;*
4. *there is a one-to-one correspondence between the suffixes of w and the leaves of STree(w), i.e., every suffix is spelled out by a unique path from the root to a leaf.*

Since any substring of w is a prefix of some suffix of w, all substrings of w can be represented as a path from the root in *STree*(w). For any node v, let *str*(v) denote the string which is a concatenation of the edge labels from the root to v. A locus of a substring x of w in *STree*(w) is a pair (v, γ) of a node v and a (possibly empty) string γ, such that *str*$(v)\gamma = x$ and γ is the shortest. A locus is said to be an explicit node if $\gamma = \varepsilon$, and is said to be an implicit node otherwise. It is well known that *STree*(w) can be represented with $O(n)$ space, by representing each edge label x with a pair (i, j) of positions satisfying $x = w[i..j]$.

Theorem 1 ([8])**.** *Given a string w of length n over an integer alphabet, STree(w) can be computed in $O(n)$ time.*

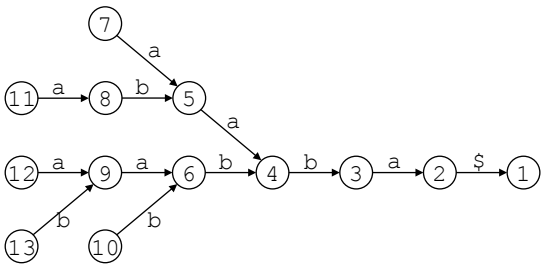

Figure 1: $CST(W)$ for $W = \{\texttt{aaba\$}, \texttt{bbba\$}, \texttt{ababa\$}, \texttt{aabba\$}, \texttt{babba\$}\}$. Each node u is associated with $id(u)$.

2.3 Suffix Trees of multiple strings

A *generalized* suffix tree of a set of strings is the suffix tree that contains all suffixes of all the strings in the set. Generalized suffix trees for a set $W = \{w_1\$, \ldots, w_k\$\}$ of strings over an integer alphabet can be constructed in linear time in the total length of the strings.

Suppose that the set W of strings is given as a *reversed trie* called a common-suffix trie, which is defined as follows.

Definition 2 (Common-suffix tries [5])**.** *The common-suffix trie of a set W of strings, denoted $CST(W)$, is a reversed trie such that*

1. *each edge is labeled with a character in Σ;*
2. *any two in-coming edges of any node are labeled with distinct characters;*
3. *each node v represents the string that is a concatenation of the edge labels in the path from v to the root;*
4. *for each string $w \in W$ there exists a unique leaf which represents w.*

An example of $CST(W)$ is illustrated in Figure 1.

Let ℓ be the number of nodes in $CST(W)$, and let $\mathit{Suffix}(W)$ be the set of suffixes of the strings in W, i.e., $\mathit{Suffix}(W) = \bigcup_{w \in W} \mathit{Suffix}(w)$. Clearly, ℓ equals to the cardinality of $\mathit{Suffix}(W)$ (including the empty string). Hence, $CST(W)$ is a natural representation of the set $\mathit{Suffix}(W)$. If L is the total length of strings in W, then $\ell \leq L + 1$ holds. On the other hand, when the strings in W share many suffixes, then $L = \Theta(\ell^2)$ (e.g., consider the set of strings $\{ab^i \mid 1 \leq i \leq \ell\}$). Therefore, $CST(W)$ can be regarded as a compact representation of the set W of strings.

Definition 3 (Suffix Trees for $CST(W)$)**.** *For any $CST(W)$, its suffix tree, denoted $STree(W)$, is a labeled rooted tree which satisfies the following:*

1. *each edge is labeled with a non-empty string which is a concatenation of the edge labels of $CST(W)$;*
2. *each internal node has at least two children;*
3. *the labels x and y of any two distinct out-going edges from the same node begin with different symbols in Σ;*
4. *there is a one-to-one correspondence between the internal nodes of $CST(W)$ and the leaves of $STree(W)$, i.e., every string which is represented by a node in $CST(W)$ is spelled out by a unique path from the root to a leaf.*

Notice that the suffix tree for $CST(W)$ is identical to a generalized suffix tree of the set W of strings. If a given $CST(W)$ is of size ℓ, then the size of the suffix tree of $CST(W)$ is $O(\ell)$.

We will use the following result in our algorithms.

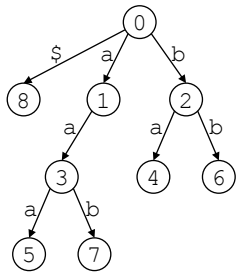

Figure 2: The LZ78 trie of abaabaaaabbaab\$. Each node numbered i represents the factor f_i of the LZ78 factorization, where f_i is the path label from the root to the node, e.g.: $f_3 = \texttt{aa}$, $f_7 = \texttt{aab}$.

Theorem 2 ([15]). *Given $CST(W)$ of size ℓ for a set W of strings over an integer alphabet, the generalized suffix tree of W can be computed in $O(\ell)$ time.*

2.4 Tools on trees

We will use the following efficient data structures on rooted trees.

Lemma 1 (Nearest marked ancestor [18, 1]). *A semi-dynamic rooted tree can be maintained in linear space so that the following operations are supported in amortized $O(1)$ time: 1) find the nearest marked ancestor of any node; 2) insert an unmarked node; 3) mark an unmarked node.*

By "semi-dynamic" above we mean that no nodes are to be deleted from the tree.

Lemma 2 (Level ancestor query [4, 3]). *Given a static rooted tree, we can preprocess the tree in linear time and space so that the ith node in the path from any node to the root can be found in $O(1)$ time for any integer $i \geq 0$, if such exists.*

3 Algorithms

In this section, we propose algorithms to compute *LZ78 factorizations* [19] and *position heaps* [7] which run in linear time for an integer alphabet.

3.1 Computing LZ78 trie from suffix tree

The LZ78 factorization [19] of a string w is a sequence $f_1, \ldots, f_m$ of non-empty substrings of w, where $f_1 \cdots f_m = w$, and each f_i is the longest prefix of $w[|f_1 \cdots f_{i-1}| + 1 \ldots n]$ such that $f_i \in \{f_j c \mid 1 \leq j < i, c \in \Sigma\} \cup \Sigma$. Each f_i is called an LZ78 factor of w. The dictionary of LZ78 factors of a string w can be represented by the following trie, called the *LZ78 trie* of w.

Definition 4. *The LZ78 trie of string w, denoted $LZ78Trie(w)$, is a rooted tree such that each node represents an LZ78 factor f_i, and there is an edge (f_j, c, f_i) with label $c \in \Sigma$ iff $f_i = f_j c$.*

See Figure 2 for an example of $LZ78Trie(w)$. $LZ78Trie(w)$ requires $O(m)$ space, where m is the number of factors in the LZ78 factorization of w. Each factor f_i can be computed in $O(|f_i|)$ time from the trie, by starting from node f_i and concatenating edge labels between f_i and the root. We compute $LZ78Trie(w)$ as a compact representation of the LZ78 factorization of w.

We present an $O(n)$-time algorithm to compute LZ78 trie of string w of length n over an integer alphabet, via the suffix tree of w. In so doing, we make use of the following key observation: since $LZ78Trie(w)$ is a trie whose nodes are all substrings of w, it can be *superimposed* on $STree(w)$, and be completely contained in it, with the exception that some nodes of the trie may correspond to implicit nodes of the suffix tree. See Figure 3 for an example.

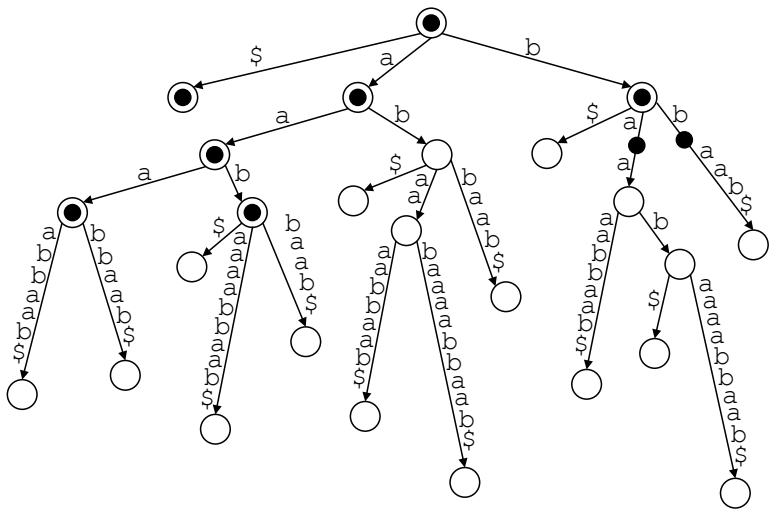

Figure 3: The LZ78-trie of string $w = \texttt{abaabaaaabbaab\$}$, superimposed on the suffix tree of w. The subtree consisting of the dark nodes is the LZ78-trie, derived from the LZ78-factorization: $\texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ba}, \texttt{aaa}, \texttt{bb}, \texttt{aab}, \$$, of w.

Theorem 3. *Given a string w of length n over an integer alphabet, LZ78Trie(w) can be constructed in $O(n)$ time and $O(n)$ working space.*

Proof. Suppose the LZ78 factorization $f_1 \cdots f_{i-1}$, up to position $p - 1 = |f_1 \cdots f_{i-1}|$ of a given string w, has been computed, and the nodes of the LZ78 trie for $f_1, \ldots, f_{i-1}$ have been added to $\mathit{STree}(w)$. Now, we wish to calculate the ith LZ78-factor f_i starting at position p. Let z be the leaf of the suffix tree that corresponds to the suffix $w[p..n]$. The longest previous factor x that is a prefix of $w[p..n]$ corresponds to the longest path of the LZ78 trie built so far, which represents a prefix of $w[p..n]$. If we consider the suffix tree as a semi-dynamic tree, where the nodes corresponding to the superimposed LZ78-trie are dynamically inserted and marked, the node x we are looking for is the *nearest marked ancestor* of z, which can be computed in $O(1)$ time. If x is not branching, then we we simply move down the edge by a single character (say a), create a new node if necessary, and mark the node representing the ith LZ78 factor $f_i = xa$. If x is branching, then we can locate the out-going edge of x that is in the path from x to the leaf z in $O(1)$ time by a level ancestor query from z. Then we insert/mark the new node for the ith LZ78 factor f_i. Technically, our suffix tree is semi-dynamic in that new nodes are created since the LZ78-trie is superimposed. However, since we are only interested in level ancestor queries at branching nodes, we only need to answer them for the original suffix tree. Therefore, we can preprocess the tree in $O(n)$ time and space to answer the level ancestor queries in $O(1)$ time. Finally, we obtain the LZ78-trie by removing the unmarked nodes from the provided suffix tree. □

3.2 Computing position heap from suffix tree

Here, we show how to compute the position heap of a set of strings from the corresponding suffix tree. We begin with the definition of position heaps.

Let $W = \{w_1\$, w_2\$, \ldots, w_k\$\}$ be a set of strings such that $w_i\$ \notin \mathit{Suffix}(w_j\$)$ for any $1 \leq i \neq j \leq k$. Define the total order $\prec$ on Σ^* by $x \prec y$ iff either $|x| < |y|$ or $|x| = |y|$ and x^R is lexicographically smaller than y^R. Let $\mathit{Suffix}_{\prec}(W)$ be the sequence of strings in $\mathit{Suffix}(W)$ that are ordered w.r.t. $\prec$ and let $\ell = |\mathit{Suffix}_{\prec}(W)|$. For any $1 \leq i \leq \ell$, let s_i denote the ith suffix of $\mathit{Suffix}_{\prec}(W)$.

Definition 5 (Position heaps for multiple strings)**.** *The* position heap *for a set W of strings, denoted $PH(W)$, is the trie heap defined as follows:*

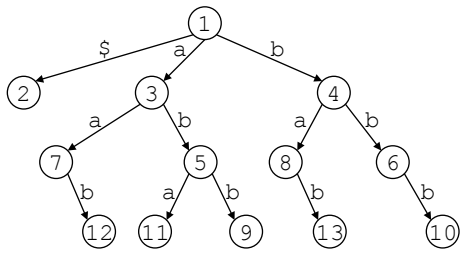

Figure 4: $PH(W)$ for $W = \{\texttt{aaba\$}, \texttt{bbba\$}, \texttt{ababa\$}, \texttt{aabba\$}, \texttt{babba\$}\}$, where $\mathit{Suffix}_{\prec}(W) = \langle \varepsilon,\ \texttt{\$},\ \texttt{a\$},\ \texttt{ba\$},\ \texttt{aba\$},\ \texttt{bba\$},\ \texttt{aaba\$},\ \texttt{baba\$},\ \texttt{abba\$},\ \texttt{bbba\$},\ \texttt{ababa\$},\ \texttt{aabba\$},\ \texttt{babba\$}\rangle$. The node labeled with integer i corresponds to s_i.

1. *each edge is labeled with a character in* Σ*;*
2. *any two out-going edges of any node are labeled with distinct characters;*
3. *the root node is labeled with* 1 *and the other nodes are labeled with integers from* 1 *to* ℓ *such that parents' labels are smaller than their children's labels;*
4. *the path from the root to node labeled with* i $(1 \leq i \leq \ell)$ *is a prefix of* s_i*.*

Notice that $PH(W)$ can be obtained by inserting, into a trie, the strings in $\mathit{Suffix}_{\prec}(W)$ in increasing order w.r.t. $\prec$. In this paper, we assume that $CST(W)$ is given as input, but $\mathit{Suffix}_{\prec}(W)$ is not explicitly given. For each s_i in $\mathit{Suffix}_{\prec}(W)$, let $id(s_i) = i$. We would like to know $id(s)$ for all suffixes s represented by $CST(W)$, which gives us the ordering of strings in $CST(W)$ w.r.t. $\mathit{Suffix}_{\prec}(W)$.

Lemma 3. *$id(s)$ for all nodes s in $CST(W)$ can be computed in $O(\ell)$ time.*

Proof. We firstly construct the suffix array of $CST(W)$ in $O(\ell)$ time, using the algorithm proposed by Ferragina et al. [9]. This gives us the lexicographical order of the suffixes represented by $CST(W)$. Secondly, we bucket-sort the nodes of $CST(W)$: we use an array of size x as buckets, where $x \leq \ell$ is the length of the longest string in $CST(W)$. We then scan the suffix array from the beginning to the end, and insert each node (string) s into the $|s|$th bucket (entry) of the array. This gives us $id(s)$ for all nodes s in $CST(W)$ in $O(\ell)$ time. □

For any $1 \leq i \leq \ell$, where ℓ is the number of nodes of $CST(W)$, let $CST(W)^i$ denote the subtree of $CST(W)$ consisting only of the nodes s_j with $1 \leq j \leq i$. $PH(W)^i$ is the position heap for $CST(W)^i$ for each $1 \leq i \leq \ell$, and in our algorithm which follows, we construct $PH(W)$ incrementally, in increasing order of i.

We present an $O(\ell)$-time algorithm to compute $PH(W)$ from the generalized suffix tree for $CST(W)$ with ℓ nodes. Since $PH(W)$ is a trie where each node represents some substring of the strings in W, it can be *superimposed* on the generalized suffix tree of W which is equivalent to the suffix tree of $CST(W)$, and be completely contained in it, with the exception that some nodes of the trie may correspond to implicit nodes of the suffix tree. See Figure 5 for an example of $PH(W)$ superimposed to the suffix tree of $CST(W)$. We summarize our algorithm as follows.

Theorem 4. *Given $CST(W)$ with ℓ nodes representing a set W of strings over an integer alphabet, $PH(W)$ can be constructed in $O(\ell)$ time and $O(\ell)$ working space.*

Proof. Suppose we have computed the position heap $PH(W)^{i-1}$ superimposed onto the suffix tree of $CST(W)$, and we wish to find the next node which corresponds to suffix s_i. Let z be the leaf of the suffix tree that corresponds to the suffix s_i. The longest prefix of s_i that is represented by $PH(W)^{i-1}$ corresponds to the longest path of $PH(W)^{i-1}$, which represents a prefix of s_i. Therefore, this can be found by a semi-dynamic nearest marked ancestor query, and the rest is analogous to the algorithm to compute the LZ78 trie of Theorem 3. Finally, we obtain the position heap by removing the unmarked nodes from the provided suffix tree. □

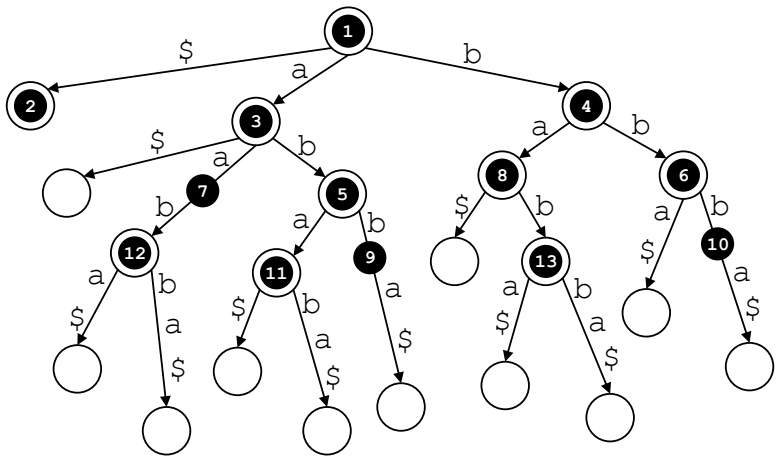

Figure 5: The position heap of set $W = \{\texttt{aaba\$}, \texttt{bbba\$}, \texttt{ababa\$}, \texttt{aabba\$}, \texttt{babba\$}\}$, superimposed on the generalized suffix tree of W, which is equivalent to the suffix tree of $CST(W)$. The subtree consisting of the dark nodes is the position heap of W.

References

[1] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[2] Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 factorization of grammar compressed text. In *Proc. SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 86–98, 2012.

[3] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[4] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[5] Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191(1–2):131–144, 1998.

[6] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.

[7] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 9(1):100–121, 2011.

[8] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. FOCS 1997*, pages 137–143, 1997.

[9] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[10] Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[11] Travis Gagie, Wing-Kai Hon, and Tsung-Han Ku. New algorithms for position heaps. In *CPM*, pages 95–106, 2013.

[12] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Linled dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*.

[13] Ming Li and Ronan Sleep. An LZ78 based string kernel. In *Proc. ADMA 2005*, pages 678–689, 2005.

[14] Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The position heap of a trie. In *Proc. SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 360–371, 2012.

[15] Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Transactions on Fundamentals of Electronics*, E86-A(5):1061–1066, 2003.

[16] Yuya Tamakoshi, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. From run length encoding to LZ78 and back again. In *Proc. DCC 2013*, pages 143–152, 2013.

[17] P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[18] Jeffery Westbrook. Fast incremental planarity testing. In *Proc. ICALP 1992*, number 623 in LNCS, pages 342–353, 1992.

[19] J. Ziv and A. Lempel. Compression of individual sequences via variable-length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.